ORACLE Academy

Using Randomization and Understanding Dot Notation and Constructors

What Will I Learn?

Objectives

- Create randomized behaviors
- Define comparison operators
- Create if-else control statements
- Create an instance of a class
- Recognize and describe dot notation

Why Learn It?

Purpose

There may be times when you are programming a game that you want your objects to move in a more realistic manner, or a less predicable way. You can use random numbers in your programming statements to achieve this behavior.

getRandomNumber Method

Use the `getRandomNumber` method to eliminate predictability in your program. The `getRandomNumber` method is a static method that:

- Belongs to the Greenfoot class
- Can be used by other classes using dot notation
- Uses a parameter that specifies the limit of the random number that will be returned
- Returns a random number between 0 (zero) and the parameter limit

Examine the `getRandomNumber` method signature:

```
public static int getRandomNumber(int limit)
```

Dot Notation

New subclasses that you create do not inherit the `getRandomNumber` method. This method must be called from the Greenfoot class using dot notation.

When a method is not in the class or inherited by the class you are programming, specify the class or object that has the method before the method name, then a dot, then the method name. This technique is called dot notation.

Example

```
Greenfoot.getRandomNumber(20);
```

Dot Notation Format

The format for dot notation code includes:

- Name of class or object to which the method belongs
- Dot
- Name of method to call
- Parameter list
- Semicolon

```
class-name.method-name (parameters);
object-name.method-name (parameters);
```

Dot Notation Example

The getRandomNumber method shown below:

- Calls a random number between 0 and up to, but not including 15.
- Returns a random number between 0 and 14.

```
Greenfoot.getRandomNumber(15)
```

Greenfoot API

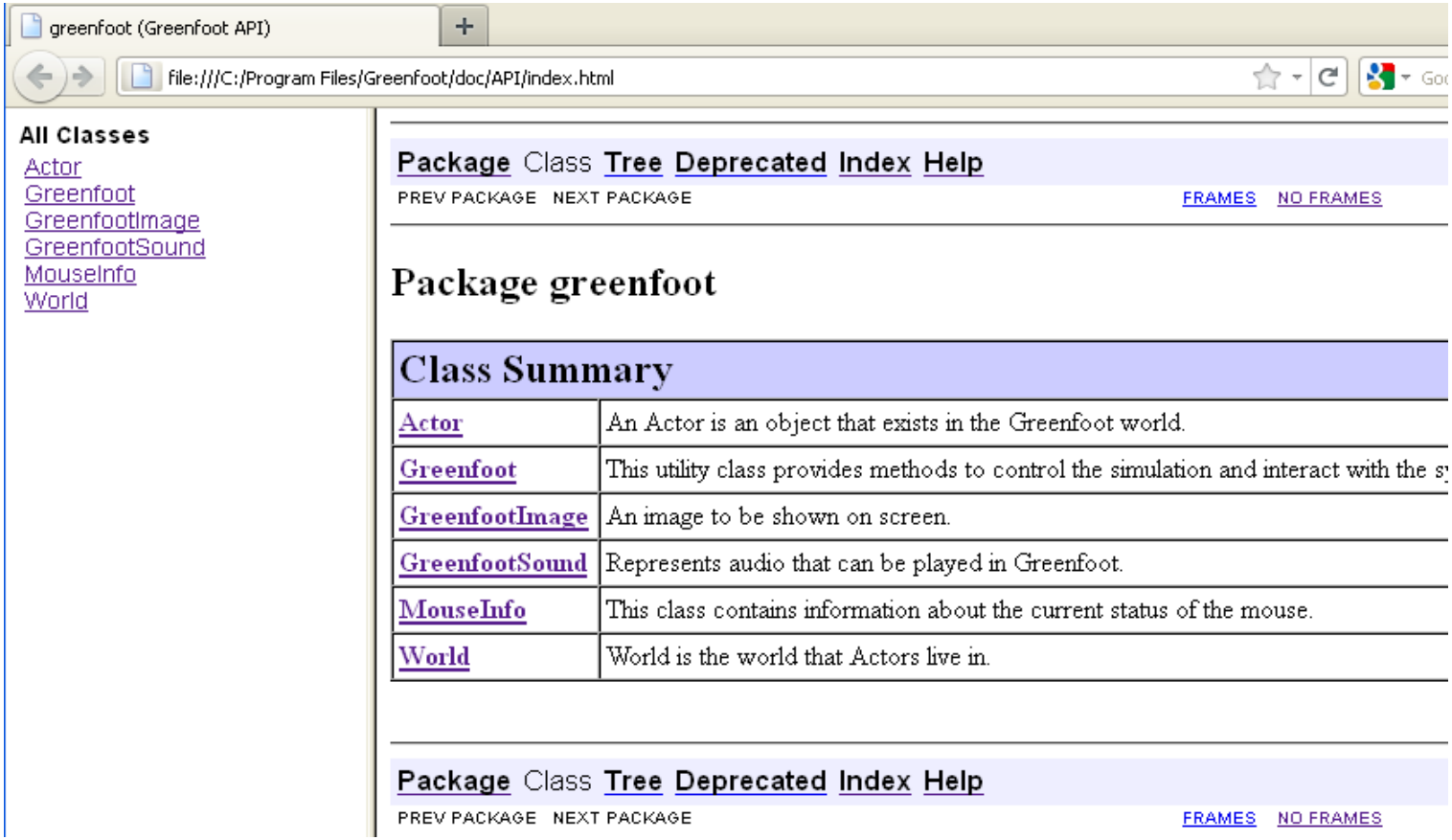
Reference the Greenfoot Application Programmers' Interface (API) to examine additional methods to call using dot notation.

The Greenfoot Application Programmers' Interface lists all of the classes and methods available in Greenfoot.

To view methods in the Greenfoot class:

1. In the environment, select the Help menu.
2. Select Greenfoot Class Documentation.
3. Click the Greenfoot class.
4. Review the method signatures and descriptions.

Greenfoot API Interface



Comparison Operators

Use comparison operators to compare a randomized value to another number in a control statement. The example below determines if the random number is less than 20 and if it is (if true) then the `turn(10);` statement is executed.

Comparison operators are symbols that compare two values.

```
If (Greenfoot.getRandomNumber(100) < 20)
{
     turn(10);
}
```

Comparison Operator Symbols

Symbol	Description
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
==	Equal
!=	Not equal

Problem Solved with Random Behavior

Problem: An object, such as a banana, should move about the game randomly so it is more challenging for the keyboard-controlled object, a monkey, to eat it.

Considerations:

- The banana should turn a small percentage of the time as it moves.
- The banana could turn a random number of degrees, up to 20 degrees, 6% of the time as it moves.

Solution expressed as a programming statement:

```
if ( Greenfoot.getRandomNumber(100) < 6)
{
     turn(Greenfoot.getRandomNumber(20) );
}
```

Random Behavior Format

The programming statement below includes:

- If-statement
- `getRandomNumber` method
  - Parameter limit (100)
  - Comparison operator (<)
  - Number to limit range of values to return (0 - 5)
- Method body with statement

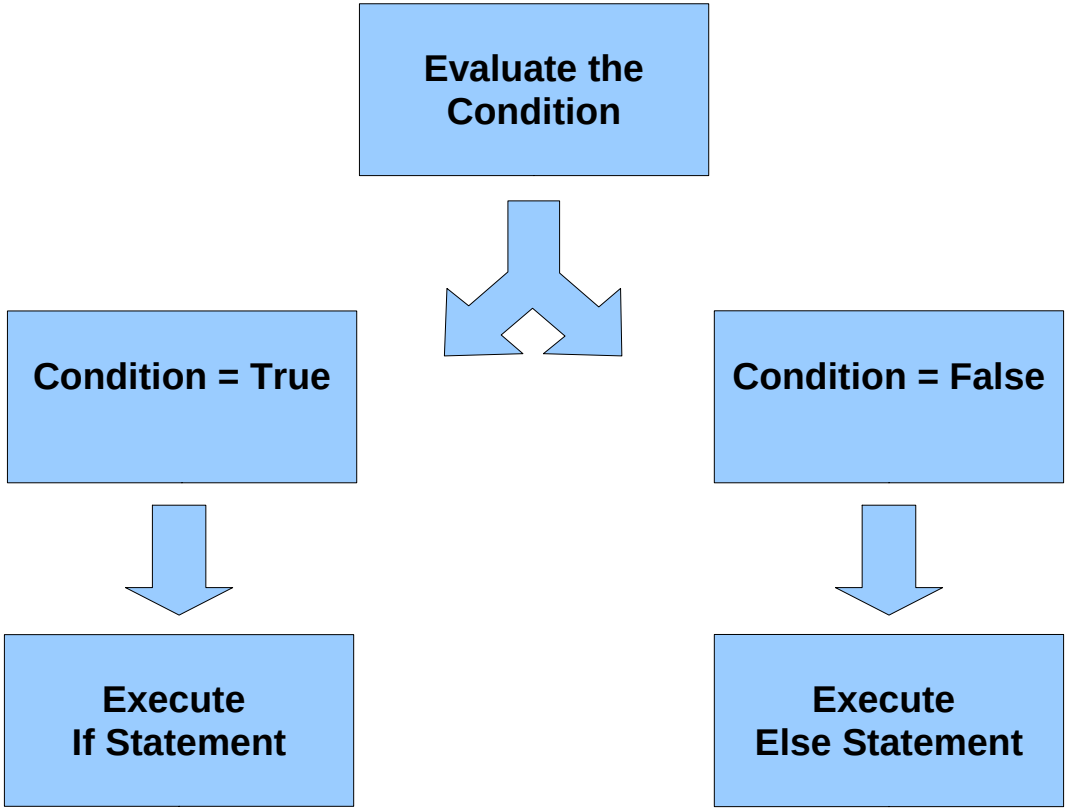
```
if ( Greenfoot.getRandomNumber(100) < 6)
{
     turn(Greenfoot.getRandomNumber(20) );
}
```

Conditional Behavior

Instances can be programmed to perform specific behaviors if a condition is not met, using an if-else statement. For example, if an instance is programmed to turn 6% of the time, what does it do the other 94% of the time?

An if-else statement executes its first code segment if a condition is true, and its second code segment if a condition is false, but not both.

If-else Statement Execution



If-else Statement Format

```
if (condition)
{
     statements;
}
else
{
     statements;
}
```

If-else Statement Example

If a random number between 0-6 is selected, turn 10 degrees. Otherwise, turn 5 degrees.

```
if (Greenfoot.getRandomNumber(100) < 7)
{
    turn(10);
}
else
{
    turn(5);
}
```

Automate Creation of Instances

Using the World subclass instance, actor instances can be programmed to automatically appear in the world when a scenario is initialized.

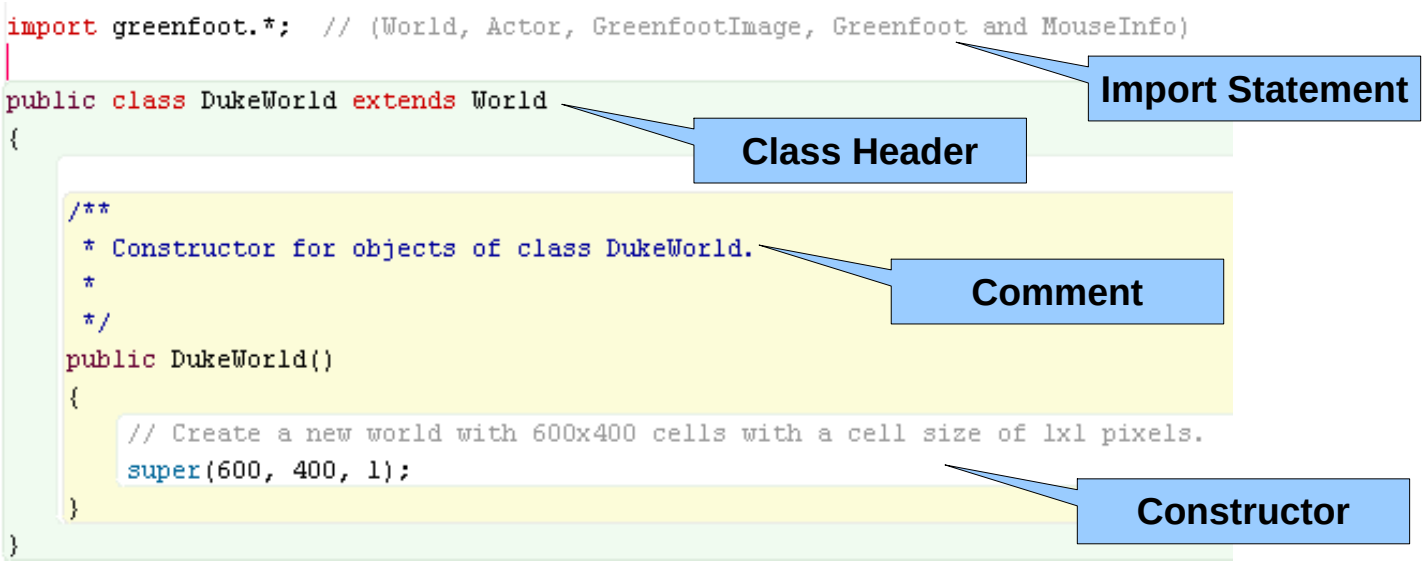
Examine the following scenario:

- Scenario: When a Greenfoot scenario (such as leaves and wombats) is started, the instances must be added by the player to play the game.
- Greenfoot default behavior:
  - The World subclass instance is automatically added to the environment after compilation or initialization of a scenario.
  - The Actor subclass instances must be manually added.
- Solution: Program instances to be automatically added to the world when the scenario is initialized.

Source Code for World Class

The World class source code includes:

- Import statement as the first line
- Class header
- Comments
- Constructor



World Constructor

Use the World constructor to automate creation of an Actor instance when the scenario is initialized.

Constructors:

- Define the instance's size and resolution
- Have no return type
- Have the same name as the name of the class (e.g., a World constructor is named World)

A constructor is a special kind of method that is automatically executed when a new instance of the class is created.

World Constructor Example

Constructor details:

- Size: x = 600, y = 400
- Resolution: 1 pixel per cell
- Keyword *super*: calls the superclass World for each instance of the DukeWorld subclass

```
public DukeWorld()
{
     super(600, 400, 1);
}
```

Size

Resolution

Automatically Create Actor Instances

To automatically create actor instances, use the `addObject` method in the World constructor. This example adds a Duke object at specified X and Y coordinates.

```
public DukeWorld()
{
     super(560, 560, 1);
     addObject (new Duke(), 150, 100);
}
```

addObject Method

The `addObject` method:

- Is a method of the World class that adds a new object to the world at specific x and y coordinates.
- Uses the keyword `new` to indicate that Greenfoot needs to create a new object of a specific class.
- Uses the following method parameters:
  - Named object from Actor class
  - Integer value of X coordinate
  - Integer value of Y coordinate

Examine the `addObject` Method signature:

```
void addObject(Actor object, int x, int y)
```

new Keyword

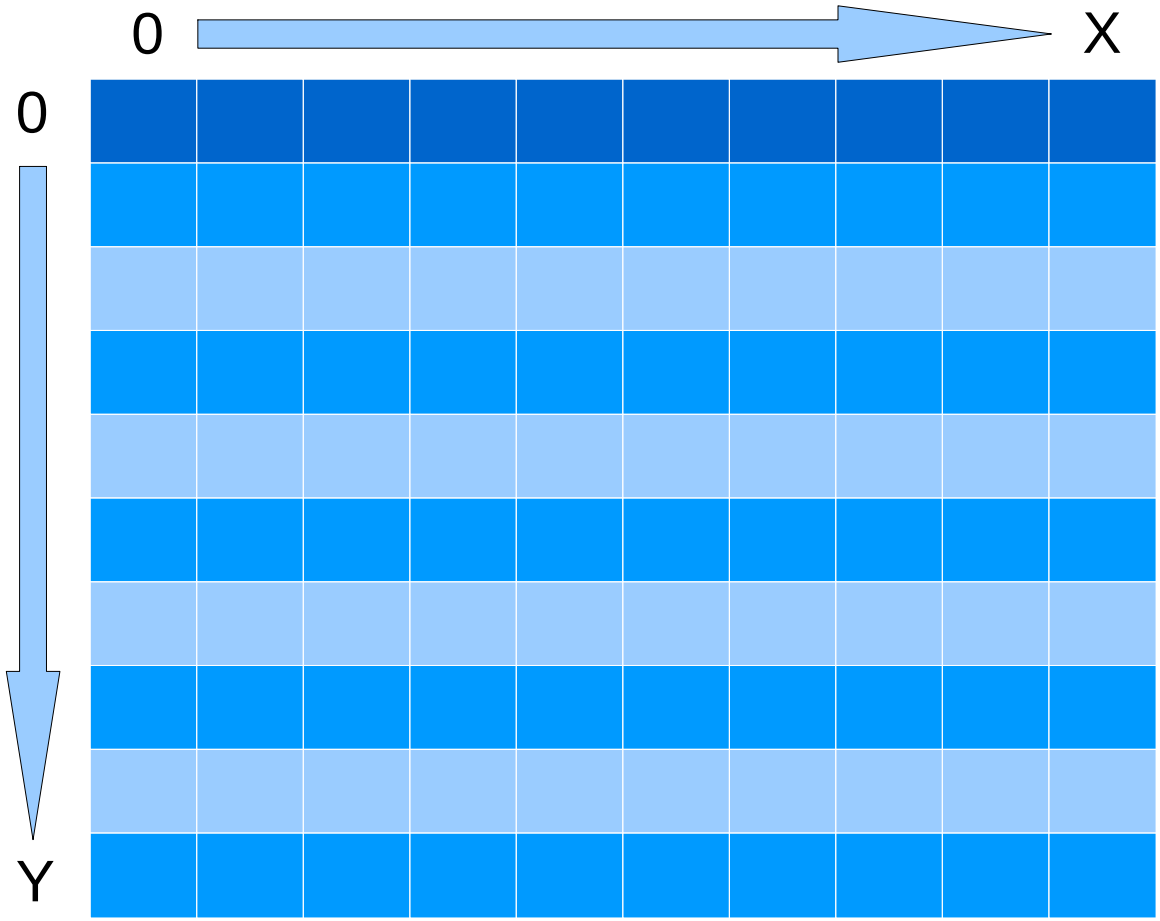
The new keyword:

- Creates new instances of existing classes.
- Starts with the keyword new, followed by the constructor to call.
  - The constructor's parameter list passes arguments (specific values) to the constructor that are needed to initialize the object's instance variables.
  - The default constructor has an empty parameter list and sets the object's instance variables to their default values.

Examine the new keyword syntax:

```
new Constructor-name()
```

Greenfoot World Coordinate System



Add Objects Using World Constructor Example

```
import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

public class DukeWorld extends World
{

    /**
     * Constructor for objects of class DukeWorld.
     *
     */
    public DukeWorld()
    {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(600, 400, 1);
        addObject(new Duke(), 150,100);
    }
}
```

Terminology

Key terms used in this lesson included:

Comparison operators

Constructor

Dot notation

new Keyword

Summary

In this lesson, you learned how to:

- Create randomized behaviors
- Define comparison operators
- Create if-else control statements
- Create an instance of a class
- Recognize and describe dot notation

Practice

The exercises for this lesson cover the following topics:

- Creating randomized actor behavior using comparison operators
- Creating instances of a class